ARL-TR-7315 ● JUNE 2015

US Army Research Laboratory

Analysis and Implementation of Particle-to-Particle (P2P) Graphics Processor Unit (GPU) Kernel for Black-Box Adaptive Fast Multipole Method

by Richard H Haney, Eric Darve, Mohammad P Ansari, Rohit Pataki, AmirHossein AminFar, and Dale Shires

ARL-TR-7315 ● JUNE 2015

US Army Research Laboratory

Analysis and Implementation of Particle-to-Particle (P2P) Graphics Processor Unit (GPU) Kernel for Black-Box Adaptive Fast Multipole Method

by Richard H Haney and Dale Shires
Computational and Information Sciences Directorate, ARL

Eric Darve, Mohammad P Ansari, Rohit Pataki, and AmirHossein AminFar
Mechanical Engineering Department, Stanford University

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* June 2015	**2. REPORT TYPE** Final	**3. DATES COVERED (From - To)** January 2015–May 2015
4. TITLE AND SUBTITLE Analysis and Implementation of Particle-to-Particle (P2P) Graphics Processor Unit (GPU) Kernel for Black-Box Adaptive Fast Multipole Method		**5a. CONTRACT NUMBER** W911NF-12-2-0019
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Richard H Haney, Eric Darve, Mohammad P Ansari, Rohit Pataki, AmirHossein AminFar, and Dale Shires		**5d. PROJECT NUMBER**
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) US Army Research Laboratory ATTN: RDRL-CIH-S Aberdeen Proving Ground, MD 21005		**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-7315
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		**10. SPONSOR/MONITOR'S ACRONYM(S)**
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Black-Box Adaptive Fast Multipole Method (bbAFMM) has been generating some interest within the high-performance computing community as a tractable solution to the well-known n-body problem. The bbAFMM approximates the n-body solution using a series of independent functions or kernels that are attractive to high-performance code development using one or more graphics processor unit (GPU) devices. This work follows the analysis and implementation of the direct interaction called particle-to-particle kernel for a shared-memory single GPU device using the Compute Unified Device Architecture, revealing a performance boost of greater than 500 times over the corresponding serial central processing unit implementation. The objective of this work is to both document the implementation of the GPU kernel and provide a better understanding of the observed performance through an algorithmic analysis that focuses on arithmetic intensity, GPU memory bandwidth, GPU peak performance, and the defined Peripheral Component Interconnect Express bandwidth.

15. SUBJECT TERMS

fast multipole method, high-performance computing, CUDA, PCIe, GPU, P2P

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Richard H Haney
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UU	20	**19b. TELEPHONE NUMBER (Include area code)** 410-278-7866

Contents

List of Figures

List of Tables

1. Introduction

The n-body problem, a well-established, potentially intractable algorithm with computational complexity of $O(n^2)$, is a critical component to many solutions in fields as varied as biology, chemistry, physics, and engineering.[1,2] Efforts to manage the computational cost of this algorithm have resulted in the development of approximations such as the Barnes-Hut Fast Multipole Method (FMM) and the Adaptive Fast Multipole Method (AFMM).[3–5] The Black-Box Adaptive Fast Multipole Method (bbAFMM) used in this work is a variant of the AFMM and follows a similar structure but without the explicit definition of multipole expansions, relying instead on the use of a Chebyshev interpolation to evaluate a continuous distribution of density at the surface of each box that encloses a set of particles.[6] The result is an algorithm that executes with a lower computational complexity of O(n) for n particles for any non-oscillatory function without sacrificing the efficacy of the final solution.[7]

The bbAFMM algorithm defines a set of 8 independent functions or kernels that are attractive to exploitation within the high-performance computing (HPC) community using the graphics processor unit (GPU).[8] This work examines the particle-to-particle (P2P) kernel using an algorithmic analysis that focuses not just on arithmetic intensity, but GPU memory bandwidth, GPU peak performance, and peripheral component interconnect express (PCIe) bandwidth to understand the potential performance benefit of using the GPU device for its definition. The resulting algorithmic analysis is then compared with the observed results of the GPU-defined P2P kernel we developed using the Compute Unified Device Architecture (CUDA).[9] A brief outline of the rest of this work follows.

The next section will discuss the background of the bbAFMM algorithm including a brief description of the 8 defined kernels emphasizing the P2P kernel that is the focus of this work. Section 3 will provide the algorithmic analysis of the P2P kernel and will include subsections on approximating floating-point operations, counting bytes as well as GPU peak, memory, and PCIe bandwidth, as they are critical in the final analysis. This section will also describe the computing environment employed in this work. Section 4 describes the actual implementation of the P2P kernel using the GPU and includes the resulting observed performance. This section will examine the observed results in relation to the previous section on algorithmic analysis. The last section presents conclusions and future work.

2. Background

The bbAFMM is an approximation of the n-body problem and structurally similar to AFMM defining 8 separate kernels that operate within a tree structure.[10] Each level $l+1$ in the tree is refined recursively from level l by subdividing cubic data structures containing source particles under consideration commonly referred to as boxes.[10,5] It is during the execution of these recursive phases that the 8 kernels are called, and can be defined as, either leaf or nonleaf modes—i.e., execute at either the leaf or nonleaf levels of the defined tree structure.

The multipole-to-multipole (M2M), multipole-to-local (M2L), particle-to-local (P2L), and local-to-local (L2L) are defined as nonleaf, while the particle-to-multipole (P2M), P2P, multipole-to-particle (M2P), and local-to-particle (L2P) are defined as leaf-mode operations. These kernels function together to approximate the global solution bbAFMM through far- and near-field computations, the latter of which is the result of the execution of the P2P kernel.

The P2P kernel calculates the contributions of particles belonging to a leaf box and its neighboring leaf boxes, and is analogous to the all-pairs interaction given by the direct n-body solution.[1,2] Letting the total number of neighboring clusters at a given level for any leaf box T be $N(T)$, the computation of near-field values using the P2P kernel is given mathematically by Eq. 1, such that x,y are defined as a well-separated pair,[11] i.e., not sharing a boundary, in target box T and source box S, respectively, for kernel $K(x,y)$ for all $x_i \in T$.[7,12]

$$f^{near}(x_i) \equiv \sum_{S \in N(T)} \sum_{y_j \in S} K(x_i y_j)\sigma_j \ . \qquad (1)$$

It follows from Eq. 1 that the GPU should be leveraged to execute the P2P kernel given the potential high floating-point count and the proclivity of the device to efficiently consume these operations.[2] However, there are other factors, which are discussed in the following section.

3. Algorithmic Analysis of P2P

This section will discuss the actual algorithmic analysis employed by this work and detail the methodology used to determine both approximate floating-point operation counts and associated bytes moved by the P2P kernel. This section also presents the actual computing environment in which this work was completed.

3.1 Computing Environment Employed

The computing environment used for this work is a 64-node heterogeneous cluster consisting of 48 IBM dx360M4 nodes, each with one Intel Phi 5110P and 16 dx360M4 nodes each with one NVIDIA Kepler K20M/K40M GPU. Each node contained dual Intel Xeon E5-2670 (Sandy Bridge) central processing units (CPUs), 64 GB of memory, and a Mellanox FDR-10 Infiniband host channel adaptor.

However, this work is focused on the behavior of a single kernel and as such does not employ multiple processors. This work makes use of a single processing core and a single NVIDIA Kepler K40 GK110 architecture with 2 PCIe Gen 3.0 slots standard, optional 2 double-widths PCIe for GPUs or coprocessors. The defined metrics for the hardware are 1) PCIe bandwidth (2 × 16 slot), 7.877 GFloat/s; Kepler K40 peak, 4,290 × 1 billion floating-point operations (GFLOPs), and 288 GB/s Kepler K40 memory bandwidth.[13,14]

3.2 Calculating FLOP and Byte Counts

The calculation of FLOPs and bytes moved by the P2P kernel follows from the algorithm employed by the CPU implementation shown in Fig. 1 for given leaf box T with "U-List" the current list of neighbors and K the defined kernel. The K kernel defined by the CPU follows the algorithm given by Fig. 2.

```
Let $S$ be the average number of particles per leaf node and $U$
be size of U-List for the cluster that the $i^{th}$ particle
resides in.
1.  FOR $i = 0$ TO $S$
2.     FOR $j = 0$ TO $U - 1$
3.        $M_j \Leftarrow (U\_List[j])$          // pointer to cluster of neighbors
4.        FOR $i2 = 0$ TO $(M_j \to SIZE)$       // total number in cluster
5.           $f[i] = f[i] + (M_j \to data[i2]).w \times K(T \to data[i], M_j \to data[i2])$
6.        END FOR
7.     END FOR
8.     FOR $i2 = i + 1$ TO $S$       // local particles
9.        $f[i] = f[i] + (T \to data[i2]).w \times K(T \to data[i], T \to data[i2])$
10.       $f[i] = f[i] + (T \to data[i]).w \times K(T \to data[i2], T \to data[i])$
11.    END FOR
```

Fig. 1 P2P algorithm

```
Input: Well-separated points $X,Y$
1.  FUNCTION K (parameters: X, Y)
2.      SET $d$ TO $\sqrt{(X.x-Y.x)^2+(X.y-Y.y)^2+(X.z-Y.z)^2}$
3.      Return $\left(1.0/d\right)$
4.  END FUNCTION
```

Fig. 2 K kernel algorithm

Given that specific FLOP counts are intrinsically bound to hardware design, we take an asymptotic approach and ignore constants, examining instead how the algorithm scales for given inputs.[15] The analysis completed by this work will focus on the execution of the P2P kernel for a single leaf node N, but these derivations can be extended without loss of generality to all leaves in the tree structure employed by bbAFMM. Analysis of the P2P algorithm in Fig. 1 reveals several nested loops that depend both on the number of neighboring particle clusters and all locally defined particles. The operations that occur within these loops compute the values for the neighboring particles f^M and local particles f^L shown in Fig. 1 lines 4–6 and 8–11, respectively.

The mathematical representation of the P2P algorithm is shown as Eq. 2 with M_j^{size} being the total number of particles defined in cluster U_List at index j.

$$\sum_{i=0}^{S}\left[\sum_{j=0}^{U-1}\left(\sum_{i2=0}^{M_j^{size}} f^M\right)+\sum_{i2=i+1}^{S} f^L\right]. \tag{2}$$

Let the number of neighboring particles from the cluster at index j shown in Fig. line 4 be averaged as Eq. 3, which assumes a worst-case scenario.

$$\frac{M_{j=0}^{size}+M_{j=1}^{size}+\ldots+M_{j=U-1}^{size}}{U-1}\approx S\,. \tag{3}$$

Following asymptotic analysis,[15] the operations f^M and f^L are defined as constants, and by applying Eq. 3, Eq. 2 becomes Eq. 4.

$$\sum_{i=0}^{S}\left[\sum_{j=0}^{U-1}S+\sum_{i2=i+1}^{S}\right]\Rightarrow\sum_{i=0}^{S}\left(\sum_{j=0}^{U-1}S\right)+\sum_{i=0}^{S}\left(\sum_{i2=i+1}^{S}\right)\Rightarrow \mathrm{O}\left(US^2+S^2\right)\Rightarrow \mathrm{O}\left(US^2\right). \tag{4}$$

The determination of the total number of bytes moved by the P2P kernel follows the algorithm presented by Fig. 1 but only counts any potential movement across the PCIe bus, the established bottleneck for many GPU-based solutions.[16,17]

Limiting the byte count to global data movements removes the K kernel from the equation and leaves only the particles defined by the given leaf box T and any neighbors defined by Eq. 3 for each member in the U-List. The resulting asymptotic behavior of bytes moved for the P2P kernel is shown as Eq. 5.

$$\mathrm{O}(US + S) \Rightarrow \mathrm{O}(US). \tag{5}$$

3.3 Kernel Arithmetic Intensity

The arithmetic intensity is the ratio of FLOPs (Eq. 4) to bytes moved (Eq. 5) and is directly related to potential GPU performance such that the higher the ratio, the better the performance is likely to be using the GPU. Clearly the arithmetic intensity given by the P2P kernel scales as a product of the average number of particles per leaf node S and the average number of particles per neighboring clusters. This computed arithmetic intensity is examined in relation to the defined GPU memory bandwidth, GPU peak performance, and PCIe bandwidth in the next subsection.

3.4 Hardware-Derived Metrics

There are 2 main issues when running a kernel on the GPU: proper utilization of the GPU and the cost of data transfer over the PCIe. Given a kernel with high arithmetic intensity such as P2P, proper utilization of the GPU is accomplished with a bigger kernel and setting the number of threads per block at optimal levels. However, estimating the cost of data transfer over the PCIe bus is more involved.

The estimated speed at which the PCIe can supply the data necessary for the P2P kernel to operate at optimal levels must be determined. The performance of the hardware employed in this work, detailed in Section 3.1, is used to derive the metrics for this subsection. The estimated PCIe bandwidth is given by Eq. 6 and the estimated GPU computational speed is given by Eq. 7, with K_{byte} and K_{flop} the number of bytes and FLOPs for the P2P kernel, and $PCIe_{bw}$ and GPU_{flop} the PCIe bandwidth and GPU FLOPs peak.

$$T_{PCIe} \equiv \frac{K_{byte} \times 10^{-9}}{PCIe_{bw}}. \tag{6}$$

$$T_{flop} \equiv \frac{K_{flop} \times 10^{-9}}{GPU_{flop}}. \tag{7}$$

Computing the values for Eqs. 6 and 7 is accomplished by factoring out U for Eqs. 4 and 5 to determine the approximate FLOPs and bytes moved for the P2P kernel. The approximated kernel FLOPs and bytes moved are then applied to Eqs. 6 and 7 for the final estimated performance results. The estimated performance results are computed for increasing numbers of particles and shown in the Table, where S is defined as the number of particles.

Table Estimated P2P kernel performance

Number of Particles	T_{PCIe}	T_{flop}	$\lvert T_{PCIe} - T_{flop} \rvert$
4,913	6.24e-07	5.63e-06	5.001e-06
9,261	1.18e-06	2.00e-05	1.88e-05
13,824	1.75e-06	4.45e-05	4.28e-05
24,389	3.10e-06	1.39e-04	1.36e-04
97,336	1.24e-05	2.21e-03	2.20e-03

The Table shows that for increasing numbers of particles, the time it takes the PCIe to transfer data to the kernel is less than it takes the device to process the results. This indicates an operation that is compute-bound rather than memory-bound and likely a good candidate for definition using the GPU.

4. Implementing P2P as a GPU Kernel

A critical aspect to building an efficient P2P GPU kernel was to keep as much data as possible on the device rather than multiple calls across the PCIe bus. This was accomplished using vectors to both store actual cluster data and provide indirection pointers to these clusters. These indirection and data clusters are briefly described in the next subsection.

4.1 Kernel Indirection and Data Clusters

The set of indirection vectors that are passed to the GPU kernel consist of the global cluster index, the associated neighbor index, starting point for neighbor, and the starting point for local called *cidx*, *uidx*, *ulist*, and *cpart_idx*, respectively (see Fig. 3). The global thread index is computed using CUDA and used to determine the current local and associated neighboring clusters with computed offsets *cpart_idx* and *ulist*, respectively. The indirections provide a means to map proper algorithm behavior to the GPU, as the device will only see data within a given thread block without regard to any real structure. The indirections properly isolate threads to defined clusters of data. These indirections for local and neighbor clusters

are then applied to retrieve the actual coordinate data held with the particles structure. The number of threads per block for the GPU kernel is defined based on the total number of coordinate particles held by the particles structure.

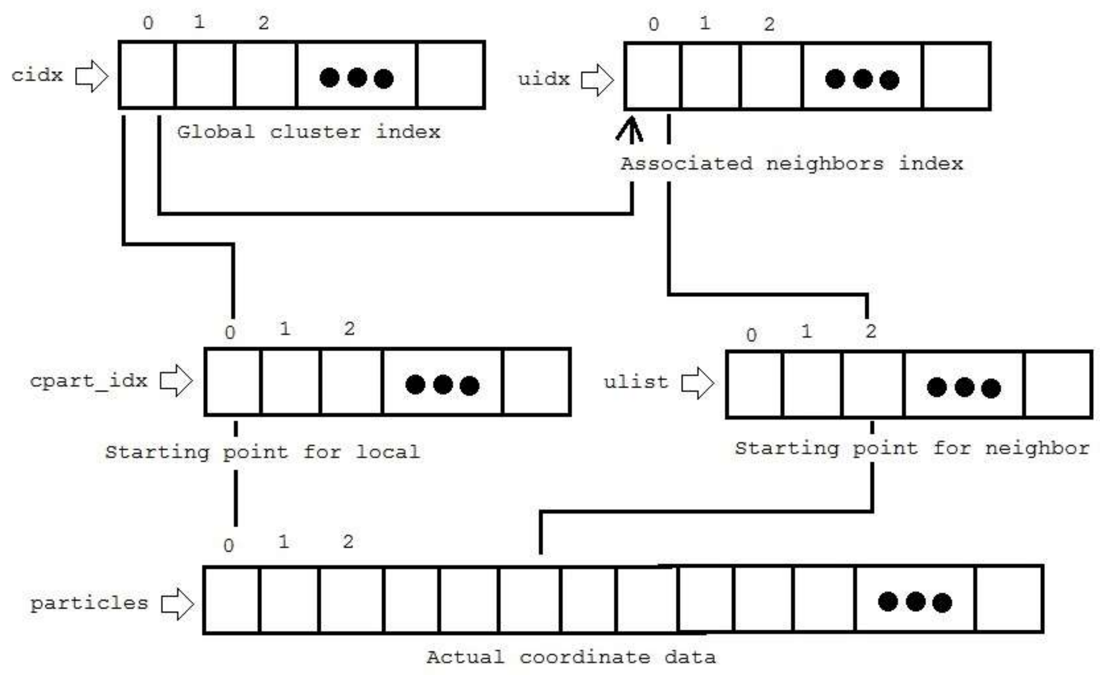

Fig. 3 P2P data and indirection pointers

The P2P GPU kernel executes for each cluster, storing the computed potentials for each in the global structure that was copied over the PCIe bus. The algorithm for the P2P GPU kernel is shown in Fig. 4. Once the kernel completes, the CPU will collect each potential from the global structure and copy it to each of the leaves in the current tree structure.

The observed performance of the implemented P2P GPU kernel and its relation to the predictive analysis are discussed in the next subsection.

```
Let $N$ be total number of particles, $P$ be collection of particles, $CIDX$ be the
global cluster index, $UIDX$ be the associated neighbors index, and $CPART\_IDX$
be the starting point to particle index
Output: $f$

1.  $gidx \Leftarrow global\_threadID$  // from CUDA
2.  IF $gidx < N$ THEN
3.          SET $F \leftarrow 0$
4.          $pi \Leftarrow P[gidx]$
5.          $I \Leftarrow CIDX[gidx]$
6.          FOR $j \leftarrow UIDX[I]$ TO $UIDX[I+1]$
7.              $J \leftarrow ULIST[j]$
8.              FOR $i2 \leftarrow CPART\_IDX[J]$ TO $CPART\_IDX[J+1]$
9.                  $pj \Leftarrow P[i2]$
10.                 $F \leftarrow F + pj.w \times K(pi, pj)$
11.             END FOR
12.         END FOR
13.         FOR $j \leftarrow CPART\_IDX[I]$ TO $gidx$
14.             $pj \Leftarrow P[j]$
15.             $F \leftarrow F + pj.w \times K(pi, pj)$
16.         END FOR
17.         FOR $j \leftarrow gidx + 1$ TO $CPART\_IDX[I+1]$
18.             $pj \Leftarrow P[j]$
19.             $F \leftarrow F + pj.w \times K(pi, pj)$
20.         END FOR
21.         $f[gidx] \leftarrow f[gidx] + F$
22. END IF
```

Fig. 4 P2P GPU kernel algorithm

4.2 Observed Performance and Predictive Analysis

The P2P GPU kernel demonstrates dramatic performance improvements over the CPU-only implementation. This GPU reveals a speed-up factor of over 500 times greater than the single-threaded CPU processor version for close to 100,000 particles, as shown in Fig. 5. These results are congruent with the predicted analysis regarding both the arithmetic intensity of the kernel itself and the hardware-derived metrics. There are several reasons for this observed performance increase when employing the GPU over the serial CPU.

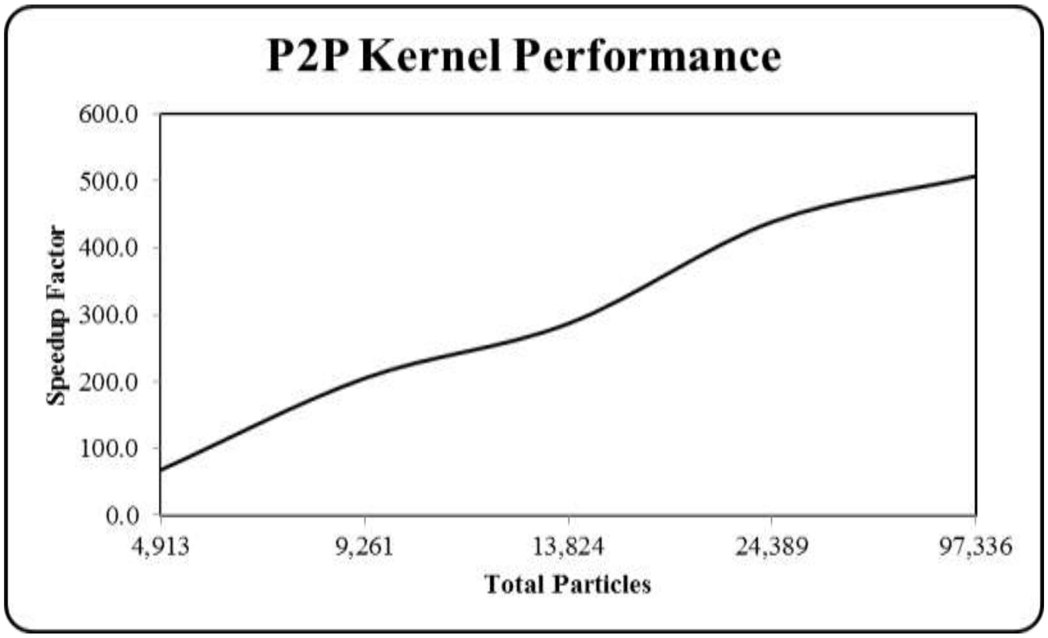

Fig. 5 GPU-defined P2P kernel performance

The first reason is the most obvious: The CPU version is not optimal given that a single process is being employed without leveraging any of the multithread capabilities. The GPU executes thousands of cores that are employed for even the simplest of functions, and this puts the CPU at a severe disadvantage from the start.[13] Another reason for the substantial performance benefit of the GPU for the P2P kernel is that both the locality and basic operation is perfectly suited for the data-throughput model of the device. The data managed by the kernel is executed in unison with no coalescing or bank conflict issues.

5. Conclusions and Future Work

This work documented the implementation of the P2P kernel for bbAFMM using a shared-memory single GPU paradigm with CUDA as the language vehicle and has shown dramatic performance increases over the corresponding CPU implementation. The P2P GPU kernel revealed a speed-up factor of more than 500 times for close to 100,000 particles. These observed results are congruent with predictive results gleaned from both algorithmic analysis and hardware-derived metrics that include GPU memory bandwidth, GPU peak performance, and PCIe bandwidth.

In the future we would like to apply these algorithm analysis techniques with other kernels defined using bbAFMM. Particular interest resides in the analysis and implementation of the M2L kernel, as this comprises the majority of bbAFMM and would present the largest payoff.

6. References

1. Ivanov L. The n-body problem throughout the computer science curriculum. Journal of Computing Sciences in Colleges, Papers of the 12th Annual Consortium for Computing Sciences (CCSC) Northeastern Conference. 2007;22(6):43–52.

2. Nyland L, Harris M, Prins J. Fast n-body simulation with CUDA. GPU Gems. 2007;3(1):677–696.

3. Greengard L, Rokhlin V. A fast adaptive multipole algorithm for particle simulations. Journal for Computational Physics. 1997;135(2):280–292.

4. Board J, Schulten K. The fast multipole algorithm. Computing in Science and Engineering. 2000;2(1):76–79.

5. Ying L, Biros G, Zorin D. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. Journal of Computational Physics. 2004;196(2):591–626.

6. Lashuk I, Chandramowlishwaran A, Langston H, Nguyen T-A, Sampath R, Shringarpure A, Vuduc R, Ying L, Zorin D, Biros G. A massively parallel adaptive fast-multipole method on heterogeneous architectures. Communications of the ACM. 2009;55(5):101–109.

7. Takahashi T, Cecka C, Darve E. Optimization of the parallel black-box fast multipole method on CUDA. InPar 2012. Proceedings of Innovative Parallel Computing 2012; 2012 May 13–14; San Jose, CA; New York (NY): Institute of Electrical and Electronics Engineers; c2012.

8. Cheng H, Greengard L, Rokhlin V. A fast adaptive multipole algorithm in three dimensions. Journal of Computational Physics. 1999;155(2):468–498.

9. NVIDIA Corporation. CUDA compute architecture: Kepler GK110. San Jose (CA): NVIDIA Corporation; 2012.

10. Ajanovic J. PCI Express* (PCIe*) 3.0 accelerator features. Santa Clara (CA): Intel Corporation; 2008.

11. Inta R, Bowman DJ, Scott SM. The “chimera”: an off-the-shelf CPU/GPGPU/FPGA hybrid computing platform. International Journal of Reconfigurable Computing: Special Issue on High-Performance Reconfigurable Computing. 2012;2012(2):10.

12. Daga M, Aji AM, Feng W-C. On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing. Presented at the 2011 Symposium on Application Accelerators in High-Performance Computing Symposium; 2011 Jul 19–20; Knoxville, TN.

13. NVIDIA. CUDA toolkit documentation. [accessed 12 April 2015]. http://docs.nvidia.com/cuda/cuda-c-programming-guide.

14. Barnes J, Hut P. A hierarchical O(N log N) force-calculation algorithm. Nature. 1986;324(6096):446–449.

15. Fong W, Darve, E. The black-box fast multipole method. Journal of Computational Physics. 2009;228(23):8712–8725.

16. Callahan PB, Kosaraju, SR. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. Journal of the ACM. 1995;42(1):67–90.

17. Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to algorithms. 2nd edition. Cambridge (MA): McGraw-Hill Book Company; 2001.

List of Symbols, Abbreviations, and Acronyms

AFMM	Adaptive Fast Multipole Method
bbAFMM	Black-Box Adaptive Fast Multipole Method
CPU	central processing unit
CUDA	Compute Unified Device Architecture
FMM	Fast Multipole Method
FLOP	floating-point operation
GFLOPs	1 billion floating-point operations
GPU	graphics processor unit
HPC	high-performance computing
L2L	local-to-local
L2P	local-to-particle
M2L	multipole-to-local
M2M	multipole-to-multipole
M2P	multipole-to-particle
P2L	particle-to-local
P2M	particle-to-multipole
P2P	particle-to-particle
PCIe	peripheral component interconnect express

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL IMAL HRA MAIL & RECORDS MGMT
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
2 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIH S R HANEY D SHIRES

INTENTIONALLY LEFT BLANK.